54325

Testing Linear Temporal Logic Formulae on Finite Execution Traces

Klaus Havelund
QSS / Recom Technologies
NASA Ames Research Center
Moffett Field, CA, 94035
havelund@ptolemy.arc.nasa.gov

Grigore Roşu
Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA, 94035
grosu@ptolemy.arc.nasa.gov

Abstract

We present an algorithm for efficiently testing Linear Temporal Logic (LTL) formulae on finite execution traces. The standard models of LTL are infinite traces, reflecting the behavior of reactive and concurrent systems which conceptually may be continuously alive. In most past applications of LTL, theorem provers and model checkers have been used to formally prove that down-scaled models satisfy such LTL specifications. Our goal is instead to use LTL for up-scaled testing of real software applications. Such tests correspond to analyzing the conformance of finite traces against LTL formulae. We first describe what it means for a finite trace to satisfy an LTL property. We then suggest an optimized algorithm based on transforming LTL formulae. The work is done using the Maude rewriting system, which turns out to provide a perfect notation and an efficient rewriting engine for performing these experiments.

1 Introduction

Linear Temporal Logic (LTL), introduced by Pnueli in 1977 [31], is a logic for specifying temporal properties about reactive and concurrent systems. The models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests, operating systems being an example. LTL has since then typically been used for specifying concurrent and interactive down-scaled models of real systems, such that fully formal program proofs could subsequently be carried out, for example using theorem provers [23, 18] or model checkers [21, 20]. However, such formal proof techniques are usually not scalable to real sized systems without an extra effort to abstract the system manually to a model which is then analyzed. Model checking of programs has received an increased attention from the formal methods community within the last couple of years . Several systems have emerged that can model check source code, such as Java, C and C++ directly (typically subsets of these languages) [22, 35, 9, 2, 25, 30]. However, these techniques will only work if abstraction is applied to the code [8, 25, 36]. Alternatives to state recording model checking have also been tried, such as VeriSoft and similar tools [13, 34], which perform stateless model checking of C++ programs, and ESC [10], which uses a combination of static analysis and theorem proving to analyze Modula3 programs and recently also Java programs. We believe these techniques will show useful for targeted verification. However, although these systems provide very high confidence in the results they provide, they scale less well. One also needs techniques that can be applied instantly and in a completely

2.1 Maude

Maude [6, 3, 4, 5] is a freely distributed high-performance system in the OBJ [16] algebraic specification family, supporting both rewriting logic [28] and membership equational logic [29]. Because of its efficient rewriting engine, able to execute up to 3 million rewriting steps per second on currently standard hardware configurations, and because of its metalanguage features based on reflection [7], Maude turns out to be an excellent tool to create executable environments for various logics, models of computation, theorem provers, and even programming languages. We were delighted to notice how easily we could implement and efficiently validate our algorithms for testing LTL formulae on finite event traces in Maude, admittedly a tedious task in C++ or Java, and hence decided to use Maude at least for the prototyping stage of our runtime check algorithms.

We very briefly and informally remind some of Maude's features, referring the interested reader to the manuals [4, 5] for more details. We'll restrict our attention to only Maude's module system and order-sorted equational logic since we don't need more for this paper. Maude supports modularization in the CLEAR [1] and OBJ [16] style of parameterized programming, with highly generic and reusable modules. There are various kinds of modules, but we are using only functional modules which follow the pattern "`fmod <name> is <body> endfm`". The body of a functional module consists of a collection of declarations, of which we are using importing, sorts, subsorts, operations, variables and equations, usually in this order. We'll describe all these "on the fly", as they appear in the paper.

2.2 Propositional Calculus

This subsection presents a decision procedure for propositional calculus due to Hsiang [26] which makes high use of associative/commutative axioms. It provides the usual truth constants (`true` and `false`) together with a potentially infinite set of propositional variables, and also the usual connectives `_/\_` (conjunction), `_++_` (exclusive disjunction), `_\/_` (disjunction), `!_` (negation), `_->_`(implication), and `_<->_`(equivalence). The procedure reduces tautology formulae to the constant `true` and all the others to some canonical form modulo associativity and commutativity.

The first algebraic specification code for this reduction procedure seems to have originally appeared in [15] in the language OBJ1, and then its OBJ3 code appeared in [16]. Below we give its obvious translation to Maude, noticing that Hsiang [26] showed that this rewriting system modulo associativity and commutativity is Church-Rosser and terminates. The Maude team was probably also inspired by this procedure, since the builtin `BOOL` module is very similar, the main difference being that `BOOL` does not allow distinguishable identifiers as boolean formulae and that the connectives are actually spelled, i.e., `_/\_` is replaced by `_and_`, `_++_` by `_xor_`, `_->_` by `_implies_`, etc.

```
fmod PROPOSITIONAL-CALCULUS is
  protecting QID .
  sort Formula .
  subsort Qid < Formula .
    *** Constructors ***
  ops true false : -> Formula .
  op _/\_ : Formula Formula -> Formula [assoc comm prec 15] .
  op _++_ : Formula Formula -> Formula [assoc comm prec 17] .
  vars X Y Z : Formula .
  eq true /\ X = X .
  eq false /\ X = false .
  eq X /\ X = X .
  eq false ++ X = X .
  eq X ++ X = false .
  eq X /\ (Y ++ Z) = X /\ Y ++ X /\ Z .
    *** Derived operators ***
  op _\/_  : Formula Formula -> Formula [assoc prec 19] .
  op !_    : Formula -> Formula [prec 13] .
  op _->_  : Formula Formula -> Formula [prec 21] .
  op _<->_ : Formula Formula -> Formula [prec 23] .
```

3 Finite Trace Linear Temporal Logic

As already explained, our goal is to develop a framework for testing software systems using temporal logic. Tests are performed on finite execution traces and we therefore need to formalize what it means for a finite trace to satisfy an LTL formula. First we present a semantics of LTL on finite traces using standard mathematical notation. Then we present a specification in Maude of a finite trace semantics. Whereas the former semantics uses universal and existential quantification, the second Maude specification is defined using recursive definitions that have a straightforward operational rewriting interpretation and which therefore can be executed.

3.1 Finite Trace Semantics

In this subsection we present a semantics of LTL on finite traces. We will regard a trace as a finite sequence of events emitted from the program that we want to observe. Such events could for example indicate when variables are written to. For example, the event *write(x,v)* would mean that "x is assigned the value v". Note that this view is slightly different from the traditional view where a trace is a sequence of program states, each state denoting the set of propositions that hold at that state. Our view is consistent with our goal to define an LTL observer as a process that is detached from the program to be observed, receiving only observed events. We shall abstract away from the concrete contents of events and just define events as a set of distinguishable identifiers. The following Maude module formalizes this idea:

```
fmod EVENT is
  protecting QID .
  sort Event .
  subsort Qid < Event .
endfm
```

It introduces the sort `Event` and states that the sort `Qid` of distinguishable identifiers is a subsort of `Event`. A trace is now a finite list of events. This is modeled by the following Maude specification:

```
fmod TRACE is
  extending EVENT .
  sort Trace .
  op end : -> Trace .
  op __ : Event Trace -> Trace [prec 25] .
endfm
```

It introduces the sort `Trace` and the constructors `end` for the empty trace, and juxtaposition of an event "e" and a trace "t", as in "$e\ t$", for creating a new trace. We shall outline the finite trace LTL semantics using standard mathematical notation rather than Maude notation. Assume two partial functions defined for nonempty traces *head* : `Trace` $\rightarrow$ `Event` and *tail* : `Trace` $\rightarrow$ `Trace` for taking the head and tail respectively of a trace, and a total function *length* returning the length of a finite trace. That is, $head(e\ t) = e$, $tail(e\ t) = t$, and $length(\texttt{end}) = 0$ and $length(e\ t) = 1 + length(t)$. Assume further for any trace t that t_i for some natural number i denotes the suffix trace that starts at position i, with positions starting at 1. The satisfaction relation $\models \subseteq$ `Trace` $\times$ `Formula` defines when a trace t satisfies a formula f, written $t \models f$, and is defined inductively over the structure of the formulae as follows, where `P` is any quoted identifier and `X` and `Y` are any formulae:

$t \models$ `P`	iff	$t \neq$ `end` and $head(t) =$ `P`,
$t \models$ `true`	iff	*true*,
$t \models$ `false`	iff	*false*,
$t \models$ `X /\ Y`	iff	$t \models$ `X` and $t \models$ `Y`,
$t \models$ `X ++ Y`	iff	$t \models$ `X` xor $t \models$ `Y`,
$t \models$ `[]X`	iff	$(\forall\ i \leq length(t))\ t_i \models$ `X`
$t \models$ `<>X`	iff	$(\exists\ i \leq length(t))\ t_i \models$ `X`
$t \models$ `X U Y`	iff	$(\exists\ i \leq length(t))$ ($t_i \models$ `Y` and $(\forall\ j < i)\ t_j \models$ `X`)
$t \models$ `o X`	iff	$t \neq$ `end` and $tail(t) \models$ `X`

```
            'a 'b 'a 'b 'a 'c 'a 'a 'b 'g 'f 'h 'c 'b 'a end .
  eq formula1 = [] ('b -> <> 'c) .
  eq formula2 = <> (! formula1) .
  eq formula3 = [] ((('a /\ o'b) \/ ('b /\ o'a)) U ('a /\ o'c)) .
endfm
```

where the three vertical dots in `trace3` stand for 100 repetitions of the previous sequence of events[4], and then try various combinations:

```
red trace1 |= formula1 .    ***> should be: false
red trace1 |= formula2 .    ***> should be: true
red trace2 |= formula1 .    ***> should be: false
red trace2 |= formula2 .    ***> should be: true
red trace3 |= formula1 .    ***> should be: false
red trace3 |= formula3 .    ***> should be: false
red trace3 |= formula2 .    ***> should be: true
```

The algorithm to test LTL formulae on traces presented above does nothing else but blindly follow the mathematical definition of satisfaction (so it is correct) and even runs reasonably fast for relatively small traces. For example, it takes less than 10,000 rewriting steps (a few milliseconds) to reduce any of the first 4 goals involving only traces of 15 events. Unfortunately this algorithm doesn't seem to be tractable for large event traces, even if run on very fast and large memory machines. That's because the number of atoms of the form `T |= X` in the boolean formula to be reduced keeps growing exponentially: besides that, the boolean reduction engine is itself intractable (it works modulo associativity and commutativity). As a practical example, it took Maude 8 million rewriting steps to reduce the fifth expression above, 53 million steps for the sixth, and it couldn't finish the last one in 10 hours.

Since the event traces generated by an executing program can easily be larger than 5,000 events, the trivial algorithm above can not be used in real practical situations.

4 An Efficient Rewriting Algorithm

In this section we shall present a more efficient rewriting semantics. First we shall motivate the design choice. Then follows the algorithm, and finally we prove that the new semantics is equivalent to the one given in the previous section.

4.1 Motivation

The operational Maude semantics of LTL that was presented in the previous section is not efficient due to the fact that the traces are carried around in several subexpressions. For example, the semantics of the until operator is given as follows:

```
eq E T |= X U Y = E T |= Y or E T |= X and T |= X U Y .
```

We can see that the trace `T` occurs in three subexpressions. A more efficient algorithm is presented below, which is based on the idea of consuming the events in the trace, one by one, and updating a data structure, say of type D, corresponding to the effect of the event on the value of the formula. Hence, we should define a function *transform* : Event $\times D \rightarrow D$. Our decision to write an operational Maude semantics this way was motivated by an attempt to program such an algorithm in Java, where such a solution would be the most natural. As it turns out, it also yields a more efficient rewriting system.

We have considered two approaches: an *automata* approach and a *formula* approach. In the automata approach one could translate the formula into an automaton, and then take the synchronized product of the

[4]The three vertical dots are not a Maude feature.

A propositional identifier is transformed to `true` if the event equals that proposition, otherwise `false`. The rule for the temporal operator `[]X` should be read as follows: the formula `X` must hold now (`X{E}`) and also in the future (`[]X`). The sub-expression `X{E}` represents the formula that must hold for the rest of the trace for `X` to hold now. As an example, consider the formula `[]<>P` where `P` is a propositional identifier. This formula applied to the distinct proposition `Q` yields the following rewritings:

```
([]<>P){Q} => []<>P /\ (<>P){Q}
           => []<>P /\ (<>P \/ P{Q})
           => []<>P /\ (<>P \/ false)
           => []<>P /\ <>P
```

As we can see, the property `<>P` has been spawned off as a consequence of the `Q` event, in addition to the original formula that still has to hold due to the "`[]`" operator.

Note that these rules spell out the semantics of each temporal operator. An alternative solution would be to define some operators in terms of others, as is typically the case in the standard semantics for LTL. For example, we could introduce an equation of the form: `<>X = true U X`, and then eliminate the rewriting rule for `<>X` in the above module. Interestingly enough this turns out to be less efficient, a result that we had not quite expected since propositional logic rewriting seems to benefit from rewriting into normal forms as demonstrated with the module `PROPOSITIONAL-CALCULUS` described in Subsection 2.2.

4.3 Revised Semantics

Before we complete the definition of our fast algorithm to evaluate formulae on finite traces, we need to introduce a new operation, `eval`, which basically "evaluates" to either `true` or `false` a formula *as it is*, that is, without using any information about the trace. This operation is needed when all the events in the trace are consumed, and basically spells out what the semantics of a formula is on an empty trace.

```
fmod EVAL is
  protecting LINEAR-TEMPORAL-LOGIC .
  op eval : Formula -> Bool .
  var P : Qid .  vars X Y : Formula .
  eq eval(P)       = false .
  eq eval(true)    = true .
  eq eval(false)   = false .
  eq eval(X /\ Y) = eval(X) and eval(Y) .
  eq eval(X ++ Y) = eval(X) xor eval(Y) .
  eq eval([] X)    = true .
  eq eval(<> X)    = eval(X) .
  eq eval(X U Y)   = eval(Y) .
  eq eval(o X)     = false .
endfm
```

The `eval` function can be seen as a morphism of logics, which maps all atomic propositions to `false`. The intuition here is that at the end of a trace, no propositions hold. The module in particular explains the semantics of the temporal operators on the empty trace. Now, the revised semantics of finite trace linear temporal logic can be implemented as follows:

```
fmod FINITE-TRACE-SEMANTICS-REVISED is
  protecting CONSUME-EVENT .
  protecting TRACE .
  protecting EVAL .
  op _ |- _ : Trace Formula -> Bool [prec 30] .
  var E : Event .  var T : Trace .  var X : Formula .
  eq end |- X = eval(X) .
  eq E T |- X = T |- X {E} .
endfm
```

```
fmod PROOF-OF-LEMMAS is
  extending FINITE-TRACE-SEMANTICS .
  extending FINITE-TRACE-SEMANTICS-REVISED .
  op e : -> Event .  op t : -> Trace .
  ops p q : -> Qid .  ops y z : -> Formula .
  eq end |= y = end |- y .
  eq end |= z = end |- z .
  eq e t |= y = t |= y {e} .
  eq e t |= z = t |= z {e} .
endfm
```

It is worth reminding the reader at this stage that the functional modules in Maude have initial semantics, so proofs by induction are valid. In particular, notice that an event can only be a specialized identifier since there are no other operations generating events. Before proceeding further, the reader should be aware of the operational semantics of the operation `_==_`, namely that the two argument terms are first reduced to their normal forms which are then compared syntactically (but modulo associativity and commutativity); it returns `true` if and only if the two normal forms are equal. Therefore, the answer `true` means that the two terms are indeed semantically equal, while `false` only means that they couldn't be proved equal; they can still be equal.

```
red (end |= p      == end |- p) and
    (end |= true   == end |- true) and
    (end |= false  == end |- false) and
    (end |= y /\ z == end |- y /\ z) and
    (end |= y ++ z == end |- y ++ z) and
    (end |= [] y   == end |- [] y) and
    (end |= <> y   == end |- <> y) and
    (end |= y U z  == end |- y U z) and
    (end |= o y    == end |- o y) and
    (p t |= p      ==   t |= p {p}) and
    (q t |= p      ==   t |= p {q}) and
    (e t |= true   ==   t |= true {e}) and
    (e t |= false  ==   t |= false {e}) and
    (e t |= y /\ z ==   t |= (y /\ z) {e}) and
    (e t |= y ++ z ==   t |= (y ++ z) {e}) and
    (e t |= [] y   ==   t |= ([] y) {e}) and
    (e t |= <> y   ==   t |= (<> y) {e}) and
    (e t |= y U z  ==   t |= (y U z) {e}) and
    (e t |= o y    ==   t |= (o y) {e}) .      ***> should be: true
```

The returned answer is indeed `true`; it took Maude 129 reductions to prove these lemmas. Notice the case analysis on the event `e` at the beginning of the second lemma's proof. Therefore, one can safely add now these lemmas as follows:

```
fmod LEMMAS is
  protecting FINITE-TRACE-SEMANTICS .
  protecting FINITE-TRACE-SEMANTICS-REVISED .
  var E : Event .  var T : Trace .  var X : Formula .
  eq end |= X = end |- X .
  eq E T |= X = T |= X {E} .
endfm
```

We can now proceed to the proof of the theorem, by induction on traces. More precisely, we show:

$\mathcal{P}$(`end`), and
$\mathcal{P}$(`T`) implies $\mathcal{P}$(`E T`), for all events `E` and traces `T`,

where $\mathcal{P}$(`T`) is the predicate "for all formulas `X`, `T |= X` iff `T |- X`". This induction schema can be easily formalized in Maude as follows:

References

[1] Rod Burstall and Joseph Goguen. The Semantics of Clear, a Specification Language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer, 1980.

[2] Tierry Cattel. Modeling and Verification of sC++ Applications. In *Proceedings of TACAS'98: Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 232–248, Lisbon, Portugal, April 1998. Springer.

[3] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. The Maude system. In Paliath Narendran and Michaël Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243, Trento, Italy, July 1999. Springer-Verlag. System Description.

[4] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and Programming in Rewriting Logic, March 1999. Maude System documentation at `http://maude.csl.sri.com/papers`.

[5] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. A Maude Tutorial, March 2000. Manuscript at `http://maude.csl.sri.com/papers`.

[6] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89. Elsevier, 1996.

[7] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.

[8] James Corbett, Matthew B. Dwyer, John Hatcliff, Corina S. Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerich, Ireland, June 2000. ACM Press.

[9] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, July 1999.

[10] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, California, USA, 1998.

[11] Doron Drusinsky. The Temporal Rover and the ATG Rover. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.

[12] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the 15th Workshop on Protocol Specification, Testing, and Verification*. North-Holland, 1995.

[13] Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, France, January 1997.

[14] Joseph Goguen, Kai Lin, Grigore Roşu, Akira Mori, and Bogdan Warinschi. An overview of the tatami project. In Kokichi Futatsugi, Tetsuo Tamai, and Ataru Nakagawa, editors, *Cafe: An Industrial-Strength Algebraic Formal Method*. Elsevier, to appear, 2000.

[15] Joseph Goguen, José Meseguer, and David Plaisted. Programming with parameterized abstract objects in OBJ. In Domenico Ferrari, Mario Bolognani, and Joseph Goguen, editors, *Theory and Practice of Software Technology*, pages 163–193. North-Holland, 1983.

[16] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.

[17] Jerry Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 331–342. Springer, 2000.

[18] Klaus Havelund. Mechanical Verication of a Garbage Collector. In José Rolim et al., editor, *Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA '99)*, volume 1586 of *Lecture Notes in Computer Science*, pages 1258–1283. Springer, 1999.